Modularity in Lattices: A Case Study on the Correspondence between Top-Down and Bottom-Up Analysis

G. Castelnuovo[1], M. Naik[2], N. Rinetzky[3], M. Sagiv[1], and H. Yang[3]

[1] Tel Aviv University [2] Georgia Institute of Technology [3] University of Oxford

Abstract. Interprocedural analyses are *compositional* when they compute over-approximations of procedures in a bottom-up fashion. These analyses are usually more scalable than top-down analyses, which compute a different procedure summary for every calling context. However, compositional analyses are rare in practice as it is difficult to develop them with enough precision.
We establish a connection between compositional analyses and *modular lattices*, which require certain associativity between the lattice join and meet operations, and use it to develop a compositional version of the connection analysis by Ghiya and Hendren. Our version is slightly more conservative than the original top-down analysis in order to meet our modularity requirement. When applied to real-world Java programs our analysis scaled much better than the original top-down version: The top-down analysis times out in the largest two of our five programs, while ours incurred only 2-5% of precision loss in the remaining programs.

1 Introduction

Scaling program analysis to large programs is an ongoing challenge for program verification. Typical programs include many relatively small procedures. Therefore, a promising direction for scalability is analyzing each procedure in isolation, using precomputed summaries for called procedures and computing a summary for the analyzed procedure. Such analyses are called *bottom-up interprocedural analysis* or *compositional analysis*. Notice that the analysis of the procedure itself need not be compositional and can be costly. Indeed, bottom-up interprocedural analyses have been found to scale well [3, 5, 11, 20, 32].

The theory of compositional analysis has been studied in [6, 10, 15, 16, 18]. However, designing and implementing such an analysis is challenging, for several reasons: it requires accounting for all potential calling contexts of a procedure in a sound and precise way; the summary of the procedures can be quite large leading to infeasible analyzers; and it may be costly to instantiate procedure summaries. An example of these challenges is the unsound original formulation of the compositional pointer analysis algorithm in [32]. A modified version of the algorithm was subsequently proposed in [29] and, more recently, proven sound in [23] using abstract interpretation. In contrast, top-down interprocedural analysis [8, 27, 30] is much better understood and has been integrated into existing tools such as SLAM [1], Soot [2], WALA [12], and Chord [26].

Our goal is to contribute to a better understanding of bottom-up interprocedural analysis. Specifically, we aimed to characterize cases under which bottom-up and top-down interprocedural analysis yield the same results when both analyses use the same

underlying abstract domains. We partially achieved our goal by formulating a sufficient condition on the effect of primitive commands on abstract states that guarantees bottom-up and top-down interprocedural analyses will yield the same results. The condition is based on lattice theory. Informally, the idea is that the abstract semantics of primitive commands can only use meet and join operations with constant elements, and that elements used in the meet must be *modular* in a lattice theoretical sense [19].

The description of the general framework and proofs of soundness and precision can be found in [4]. For space reasons, we do not provide the general theory here. Instead, we present our results by means of an application: We present a variant of *connection analysis* [14] which we developed using our approach. Connection analysis is a kind of pointer analysis that aims to prove that two references can never point to the same undirected heap component. It thus ignores the direction of pointers. This problem arose from the need to automatically parallelize sequential programs. Despite its conceptual simplicity, connection analysis is flow- and context-sensitive, and the effect of program statements is non-distributive. In fact, the top-down interprocedural connection analysis is exponential; indeed our experiments indicate that this analysis scales poorly.

More specifically, in this paper, we present a formulation of a variant of the connection analysis in a way that satisfies the requirements of our general framework. Intuitively, the main difference from the original analysis is that we had to over-approximate the treatment of variables that point to null in all program states that occur at a program point. We implemented two versions of the top-down interprocedural connection analysis for Java programs in order to measure the extra loss of precision of our over-approximation. We also implemented the bottom-up interprocedural analysis for Java programs. We report empirical results for five benchmarks of sizes 15K–310K bytecodes for a total of 800K bytecodes. The original top-down analysis times out in over six hours on the largest two benchmarks. For the remaining three benchmarks, only 2-5% of precision was lost by our bottom-up analysis due to the modularity requirement.

2 Informal Overview and Running Example

In this section, we illustrate the use of *modular lattices* in the design of our compositional connection analysis, focusing on the use of *modular elements* to ensure precision.

Definition 1. *An element d_p in a lattice $\mathcal{D}$ is* (right) modular *if*

$$\forall d, d' \in \mathcal{D}.\ d' \sqsubseteq d_p \Rightarrow d_p \sqcap (d \sqcup d') = (d_p \sqcap d) \sqcup d' \ .$$

We call a lattice $\mathcal{D}$ modular *if all of its elements are modular.*

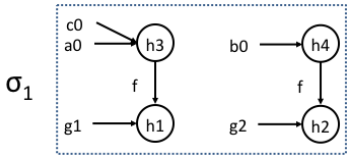
One way to understand the modularity condition in lattices is to think about it as a requirement of commutativity between two operators $(d_p \sqcap -)$ and $(- \sqcup d')$ [19]. A particular case in which this condition holds for every element d_p is if the lattice meet operation distributes over the join operation, e.g., as in the powerset lattice $(\mathcal{P}(S), \subseteq)$.

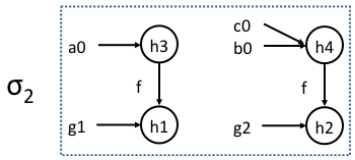
Consider a program in Fig. 1. It consists of procedures `main`() and $\mathtt{p}_1(), \ldots, \mathtt{p}_n()$. The `main`() procedure first allocates four objects and connects them into two disjoint pairs. Then, it invokes $\mathtt{p}_1()$ using either $\mathtt{a}_0$ or $\mathtt{b}_0$ as the actual parameter. This invocation triggers subsequent calls to $\mathtt{p}_2(), \ldots, \mathtt{p}_n()$, where all the invoked procedures behave almost the same as $\mathtt{p}_1$: procedure $\mathtt{p}_i()$ assigns its formal parameter $\mathtt{c}_{i-1}$ either to $\mathtt{a}_i$ or to $\mathtt{b}_i$, and then calls $\mathtt{p}_{i+1}$ using $\mathtt{c}_{i-1}$ as the actual parameter, unless $i = n$.

```
static main() {
  g1=new h1; g2=new h2;
  a0=new h3; b0=new h4;
  a0.f=g1; b0.f=g2;
  if(*) p1(a0);
  else p1(b0);}
```

```
pi(ci-1) {
  if(*)
    ai=ci-1;
  else
    bi=ci-1;
  pi+1(ci-1);}
```

```
pn(cn-1) {
  if(*)
    an=cn-1;
  else
    bn=cn-1;}
```





$d_1 = \{\{g_1, a_0, c_0\}, \{g_2, b_0\}, \{a_1\}, \{b_1\}, \ldots, \{a_n\}, \{b_n\}\}$

$d_2 = \{\{g_1, a_0\}, \{g_2, b_0, c_0\}, \{a_1\}, \{b_1\}, \ldots, \{a_n\}, \{b_n\}\}$

Fig. 1. First row: example program. All of $g_1, g_2, a_1, \ldots, b_1 \ldots$ are global variables. Second row: the concrete states at the entry of $p_1()$ and the corresponding connection abstractions.

We say that two heap objects are *connected* in a state when it is possible to reach from one object to the other, following paths in the heap ignoring pointer direction. Two variables are *connected* when they point to connected heap objects. Connection analysis soundly estimates the connection relationships between variables. The abstract states d of the analysis are families $\{X_i\}_{i \in I}$ of disjoint sets of variables ordered by refinement: Two variables x, y are in the same set X_i, which we call a *connection set*, when x and y may be connected, and $d_1 \sqsubseteq d_2 \iff \forall X_1 \in d_1. \exists X_2 \in D_2. X_1 \subseteq X_2$.

Example 1. Fig. 1 depicts the two possible concrete states, σ_1 and σ_2, that can occur at the entry to $p_1()$, and their respective connection abstractions. In the concrete states, variables $a_1, \ldots, b_1 \ldots$ point to `null`. Hence, they are represented by separate connection sets. The abstract states $\alpha(\{\sigma_1\})$ and $\alpha(\{\sigma_2\})$ are incomparable. However, both are more precise than the abstraction of a state in which all the variables point to the same object and less precise than that of state where the values of all the variables is null.

A standard approach for an interprocedural analysis is to follow the execution flow of a program top-down (i.e., from callers to callees), and to re-analyze each procedure for every different calling context. This approach often suffers from the scalability issue. One reason is the explosion of different calling contexts. Indeed, note that in our example program, each procedure $p_i()$ calls the procedure $p_{i+1}()$ with two different calling contexts. As a result, a top-down connection analysis, e.g., [14], computes 2^i abstract states at the entry to procedure $p_i()$.

Example 2. The abstract state d_t shown below arises at the entry to $p_n()$ when the then-branch is always selected, while d_e arises when only $p_{n-1}()$ selects the else-branch.

$d_t = \{\{g_1, a_0, a_1, \ldots, a_{n-2}, a_{n-1}, c_{n-1}\}, \{g_2, b_0\}, \{a_n\}, \{b_1\}, \ldots, \{b_{n-1}\}, \{b_n\}\}$

$d_e = \{\{g_1, a_0, a_1, \ldots, a_{n-2}, b_{n-1}, c_{n-1}\}, \{g_2, b_0\}, \{a_{n-1}\}, \{a_n\}, \{b_1\}, \ldots, \{b_{n-2}\}, \{b_n\}\}$

The bottom-up (compositional) approach avoids the explosion of the calling contexts that occur in the top-down analysis. It does so by analyzing each procedure independently to compute a summary, which is then instantiated as a function of a calling

context. Unfortunately, it is rather difficult to analyze a procedure independently of its calling contexts and at the same time compute a summary that is sound and precise enough. One of the reasons is that the abstract transformers may depend on the input abstract state, which is often unavailable for the compositional analysis.

We formulate a precise compositional connection analysis. The key feature of the analysis is that the abstract transformers of primitive commands a have the form

$$[\![a]\!]^\sharp = \lambda d.\, (d \sqcap d_p) \sqcup d_g \ ,$$

where d_p and d_g are some constant abstract states, independent of the input, and d_p is a modular element in the lattice of all abstract states. For example, the abstract transformer of the statement $\mathtt{x} = \mathtt{y}$ has the form above with $d_p = S_{x'}$ and $d_g = U_{x'y'}$, where $S_{x'}$ consists of two connection sets, $\{x'\}$ and the set of all the other variables, and $U_{x'y'}$ has the set $\{x', y'\}$ of x', y' and the singleton sets $\{z\}$ for all variables z other than x', y'. Intuitively, taking the meet with $S_{x'}$ separates out the variable x from its connection set in d, and joining the result with $U_{x'y'}$ adds x to the connection set of y.[1]

Note that the abstract domain is not distributive.[2] However it does contain modular elements. In particular, d_p in the abstract transfer for a above is modular. This implies that for all $d, d' \in \mathcal{D}$ such that $d' \sqsubseteq d_p$,

$$[\![a]\!]^\sharp(d \sqcup d') = d_p \sqcap (d \sqcup d') \sqcup d_g = (d_p \sqcap d) \sqcup d' \sqcup d_g = [\![a]\!]^\sharp(d) \sqcup d'$$

where the modularity is used in the second equality. Intuitively, $[\![a]\!]^\sharp(d \sqcup d')$ represents the computation of the top-down analysis, and $[\![a]\!]^\sharp(d) \sqcup d'$ that of the bottom-up analysis. In the former case, the analysis of a uses all the information available in the input abstract state $d \sqcup d'$, whereas in the latter case, the analysis ignores the additional information recorded in d' and just keeps d' in its outcome using the join operation. The equality between $[\![a]\!]^\sharp(d \sqcup d')$ and $[\![a]\!]^\sharp(d) \sqcup d'$ means that both approaches lead to the same outcome, as long as $d' \sqsubseteq d_p$ holds. This equality is the basis of our coincidence result between top-down and bottom-up analyses.

Concretely, consider the case that a is the assignment $\mathtt{a_i{=}c_{i-1}}$ in the body of the procedure $\mathtt{p_i}$. Let $\{d_k\}_k$ be all the abstract states at the entry of $\mathtt{p_i}$ encountered during the top-down analysis. Suppose that there exists d such that $\forall k : \exists d'_k \sqsubseteq S_{\mathtt{a'_i}} : d_k = d \sqcup d'_k$. Our compositional approach analyzes $\mathtt{a_i = c_{i-1}}$ only once with the abstract state d, and computes $d' = [\![\mathtt{a_i = c_{i-1}}]\!]^\sharp(d)$. Later when $\mathtt{p_i}$ gets called with d_k's, the analysis adapts d' by simply joining it with d'_k, and returns this outcome of this adaption as a result. This adaptation of the bottom-up approach gives the same result as the top-down approach, which applies $[\![\mathtt{a_i = c_{i-1}}]\!]^\sharp$ on d_k directly:

$$\begin{aligned}[\![\mathtt{a_i{=}c_{i-1}}]\!]^\sharp(d_k) &= (d_k \sqcap S_{\mathtt{a'_i}}) \sqcup U_{\mathtt{a'_i c'_{i-1}}} &&= ((d \sqcup d'_k) \sqcap S_{\mathtt{a'_i}}) \sqcup U_{\mathtt{a'_i c'_{i-1}}} \\ &= (d \sqcap S_{\mathtt{a'_i}}) \sqcup d'_k \sqcup U_{\mathtt{a'_i c'_{i-1}}} &&= [\![\mathtt{a_i{=}c_{i-1}}]\!]^\sharp(d) \sqcup d'_k \ .\end{aligned}$$

The third equality holds due to the modularity property.

[1] The subscripts p and g are used as mnemonics: d_p is used to partition a connection set and d_g to group two connection sets together.

[2] For example, let $d_1 = \{\{x, z\}, \{y\}\}$, $d_2 = \{\{x, y\}, \{z\}\}$, and $d_3 = \{\{y, z\}, \{x\}\}$). Then $d_1 \sqcap (d_2 \sqcup d_3) = d_1 \sqcap \{\{x, y, z\}\} = d_1$, but $(d_1 \sqcap d_2) \sqcup (d_1 \sqcap d_3) = \{\{x\}, \{y\}, \{z\}\}$.

3 Programming Language

We formalize our results for a simple imperative procedural programming language.

$$\begin{array}{rl} \textit{Primitive commands} & a ::= \texttt{x} = \texttt{null} \mid \texttt{x} = \texttt{new} \mid \texttt{x.f} = \texttt{y} \mid \texttt{x} = \texttt{y} \mid \texttt{x} = \texttt{y.f} \\ \textit{Commands} & C ::= \texttt{skip} \mid a \mid C; C \mid C + C \mid C^* \mid p() \\ \textit{Declarations} & D ::= \mathsf{proc}\ p() = \{\mathsf{var}\ \boldsymbol{x}; C\} \\ \textit{Programs} & Pr ::= \mathsf{var}\ \boldsymbol{g}; C \mid D; Pr \end{array}$$

We denote by PComm, G, L, and PName the sets of primitive commands, global variables, local variables, and procedure names, respectively. We use the following symbols to range over these sets: $a \in \mathsf{PComm}$, $g \in \mathsf{G}$, $x, y, z \in \mathsf{G} \cup \mathsf{L}$, and $p \in \mathsf{PName}$. We assume that L and G are fixed arbitrary finite sets. Also, we consider only well-defined programs where all the called procedures are defined.

Syntax. A program Pr in our language is a sequence of procedure declarations, followed by a sequence of declarations of global variables and a main command. Commands contain primitive commands $a \in \mathsf{PComm}$, sequential composition $C; C'$, nondeterministic choice $C + C'$, iteration C^*, and procedure calls $p()$. We use $+$ and * instead of conditionals and while loops for theoretical simplicity: given appropriate primitive commands, conditionals and loops can be easily defined. We use the standard primitive commands for pointer programs.

Declarations D give the definitions of procedures. A procedure p is comprised of a sequence of local variables declarations $\boldsymbol{x}$ and a command, denoted by C_{body_p}, which we refer to as procedure p's *body*. Procedures do not take any parameters or return any values explicitly; values can instead be passed to and from procedures using global variables. To simplify presentation, we do not consider mutually recursive procedures in our language; direct recursion is allowed.

Operational Semantics. A state $\sigma = \langle s_g, s_l, h \rangle$ is a triplet comprised of a global environment s_g, a local environment s_l and an heap h mapping locations and field names to values. For simplicity, values are either locations in the heap or the special value `null`. We say that locations o_1 and o_2 are *connected* in heap h, denoted by $o_1 \leftrightsquigarrow_h o_2$, if there exists an *undirected* path of pointer fields between o_1 and o_2. We use a relational (input-output tracking) store-based large step operational semantics which manipulates pairs of states $\langle \bar{\sigma}, \sigma' \rangle$: $\bar{\sigma}$ records the state of the program at the entry to the active procedure and σ' records the current state. For further details see [4].

4 Intraprocedural Connection Analysis

We first show how the modular elements can help in *intra*procedural analysis. For simplicity, we use in this section a non-relational semantics, i.e., the semantics only tracks the current state. In §5, we adapt the analysis to abstract the relational semantics.

Partition Domains. We first define a general notion of *partition domains*, and then instantiate it to an abstract domain suitable for connection analysis of programs without procedures. (§5 defines the general setup.) We denote by $\mathsf{Equiv}(\Upsilon) \subseteq \mathcal{P}(\Upsilon)$ the set of equivalence relations over a set Υ, ranged over by metavariable d. We use $v_1 \cong_d v_2$ to denote that $\langle v_1, v_2 \rangle \in d$, and $[v]_d$ to denote the equivalence class of $v \in \Upsilon$ induced by d. We omit the d subscript when it is clear from context. $(d_1 \cup d_2)^+$ denotes the transitive closure of $d_1 \cup d_2$. By abuse of notation, we sometimes treat an equivalence relation d as the partitioning $\{[v]_d \mid v \in \Upsilon\}$ of Υ into equivalence classes it induces.

$$[\![x = \texttt{null}]\!]^\sharp(d) = d \sqcap S_{x'} \qquad [\![x = \texttt{new}]\!]^\sharp(d) = d \sqcap S_{x'} \qquad [\![x.f = y]\!]^\sharp(d) = d \sqcup U_{x'y'}$$
$$[\![x = y]\!]^\sharp(d) = (d \sqcap S_{x'}) \sqcup U_{x'y'} \qquad [\![x = y.f]\!]^\sharp(d) = (d \sqcap S_x) \sqcup U_{x'y'}$$
$$\text{where } U_{x'y'} = \{\{x', y'\}\} \cup \{\{z'\} \mid z' \in \Upsilon \setminus \{x', y'\}\},\ S_{x'} = \{\{x'\}\} \cup \{\{z' \mid z' \in \Upsilon \setminus \{x'\}\}\}$$

Fig. 2. Abstract transfer functions for primitive commands in the connection analysis where $d \neq \bot$. For $d = \bot$, the transfer function of any primitive command $a \in \mathsf{PComm}$ is $[\![a]\!]^\sharp(d) = d$.

Definition 2. *The* partition lattice $\mathcal{D}_{\mathsf{part}}(\Upsilon)$ *over a set* Υ *is a 6-tuple*

$$\mathcal{D}_{\mathsf{part}}(\Upsilon) = \langle\, \mathsf{Equiv}(\Upsilon), \sqsubseteq, \bot_{\mathsf{part}} = \{\{a\} \mid a \in \Upsilon\}, \top_{\mathsf{part}} = \{\Upsilon\}, \sqcup = (- \cup -)^+, \sqcap = - \cap - \,\rangle$$
where $d_1 \sqsubseteq d_2 \Leftrightarrow \forall v_1, v_2 \in \Upsilon, v_1 \cong_{d_1} v_2 \Rightarrow v_1 \cong_{d_2} v_2$.

The connection abstract domain $\mathcal{D}(\Upsilon)$ *is an extension of the partition domain* $\mathsf{Equiv}(\Upsilon)$ *to include a bottom element* $\bot$ *in its carrier set, i.e.,* $\bot \sqsubset d$ *for every* $\bot \neq d \in \mathcal{D}(\Upsilon)$*, with the lattice operations extended in the obvious way. We refer to the equivalence class* $[x]_d$ *of* $x \in \Upsilon$ *as the* connection set *of* x *in* $\bot \neq d \in \mathcal{D}(\Upsilon)$.

The connection abstract domain $\mathcal{D}(\Upsilon)$ is parametrized by a set Υ pertaining to the set $\mathsf{G} \cup \mathsf{L}$ of pointer variables. For example, in the intraprocedural settings we use $\Upsilon = \{x' \mid x \in \mathsf{G} \cup \mathsf{L}\}$. Intuitively, x' and y' belong to different partitions in an abstract state $d \in \mathcal{D}(\Upsilon)$ that arises as at a program point pt during the analysis if the pointer variables `x` and `y` never point to connected heap objects when the execution of the program reaches pt. For instance, if there is a program state occurring at pt in which `x.f` and `y` point to the same heap object, then it must be that x' and y' belong to the same connection set in d. In the following, we omit Υ when it is clear from context. More formally, the abstraction map α is defined as follows: $\alpha(\emptyset) = \bot$ and $\alpha(S) = \{[x]_{d_S} \mid x \in \mathsf{G} \cup \mathsf{L}\}$ for any other set of states, where d_S is the reflective transitive closure of the relation $\bigcup_{\sigma \in S}\{(x, y) \mid \{x, y\} \subseteq \mathsf{G} \cup \mathsf{L}, \sigma = \langle s_g, s_l, h\rangle$, and $(s_g \cup s_l)x \leftrightsquigarrow_h (s_g \cup s_l)y\}$.

Abstract Semantics. The abstract semantics of primitive commands is defined in Fig. 2 using meet and join operations with constant elements to conform with the requirement of Def. 3. (Note that, as expected, the functions are strict, i.e., they map $\bot$ to $\bot$.)

Assigning `null` or a fresh object to a variable `x` separates x' from its connection set. Therefore, the analysis takes the meet of the current abstract state with $S_{x'}$ — the partition with two connection sets $\{x'\}$ and the rest of the variables. The concrete semantics of `x.f = y` redirects the `f`-field of the object pointed to by `x` to the object pointed to by `y`. The abstract semantics treats this statement quite conservatively, performing "weak updates": It merges the connection sets of x' and y' by computing the least upper bound of the current abstract state with $U_{x'y'}$ — a partition with $\{x', y'\}$ as a connection set and singleton connection sets for the other variables. The effect of the statement `x = y` is to separate the variable x' from its connection set and to add x' to the connection set of y'. This is realized by performing a meet of the current abstract state with $S_{x'}$, and then joining the result with $U_{x'y'}$. Following [14], the effect of the assignment `x = y.f` is handled in a very conservative manner, treating y and $y.f$ in the same connection set since the abstraction does not distinguish between the objects pointed to by `y` and `y.f`. Thus, the same abstract semantics is used for `x = y.f` and `x = y`.

The transformers defined in Fig. 2 are in fact the best transformers [9]: For every abstract state d, there exists a concrete state σ which has an object o_X for every partition

X in d which is pointed to by all the variables in X and has a pointer field pointing to itself. It easy to verify that it holds that $\alpha(\sigma) = d$ and $\alpha([\![a]\!](\sigma)) = [\![a]\!]^\sharp(\alpha(\sigma))$, where $[\![a]\!]$ is the concrete operational semantics of command a. The abstract semantics of composite commands is standard, and omitted [4].

Conditionally Compositional Intraprocedural Analysis. In the following we show that under certain restrictions it is possible to utilize the modularity property to compute summaries of intraprocedural commands.

Definition 3. *A function $f : \mathcal{D} \to \mathcal{D}$ is* conditionally adaptable *if $f(\bot) = \bot$ and for every $d \neq \bot$, $f(d) = (d \sqcap d_p) \sqcup d_g$ for some $d_p, d_g \in \mathcal{D}$ and the element d_p is modular. We refer to d_p as f's* meet element *and to d_g as f's* join element.

Lemma 1. *All the abstract transfer functions of the primitive commands in the intraprocedural connection analysis (shown in Fig. 2) are conditionally adaptable.*

We denote the meet and join elements of the abstract transformer $[\![a]\!]^\sharp$ of a primitive command a by $P[\![a]\!]^\sharp$ and $G[\![a]\!]^\sharp$, respectively. For a command C, we denote by $P[\![C]\!]^\sharp$ the set of the meet elements of the primitive commands occurring in C.

Lemma 2. *Let C be a command composed of primitive commands whose transfer functions are conditionally adaptable and which does not contain procedure calls. For every $d_1, d_2 \in \mathcal{D}$ such that $d_1 \neq \bot$, if $d_2 \sqsubseteq \bigsqcap P[\![C]\!]^\sharp$ then*

$$[\![C]\!]^\sharp(d_1 \sqcup d_2) = [\![C]\!]^\sharp(d_1) \sqcup d_2.$$

Intraprocedural Summaries. Lem. 2 can justify the use of compositional summaries in intraprocedural analyses in certain conditions: Take a command C and an abstract value d_2 such that the conditions of the lemma hold. An analysis that needs to compute the abstract value $[\![C]\!]^\sharp(d_1 \sqcup d_2)$ can do so by computing $d = [\![C]\!]^\sharp(d_1)$, possibly caching (d_1, d) in a summary for C, and then adapting the result by joining d with d_2.

Lem. 1 and 2 allow only for *conditional* intraprocedural summaries to be used in the connection analysis; a summary for a command C can be used only when $d_2 \sqsubseteq d_p$ for all $d_p \in P[\![C]\!]^\sharp$. In contrast, and perhaps counter-intuitively, the interprocedural analysis has non-conditional summaries, which do not have a proviso like $d_2 \sqsubseteq P[\![C]\!]^\sharp$. It achieves this by requiring certain properties of the abstract domain used to record procedures summaries, which we now describe.

5 Interprocedural Connection Analysis

In this section we define top-down and bottom-up interprocedural connection analyses, and prove that their results coincide. The main message of this section is that we can summarize the effects of procedures in a bottom-up manner, and use the modularity property to prove that the results of the bottom-up and top-down analyses coincide. This coincidence, together with the soundness of the top-down analysis (Lem. 3), ensures the soundness of the bottom-up analysis.[3]

[3] We analyze recursive procedures in a standard way using a fixpoint computation. As a result, a recursive procedure might be analyzed more than once. However, and unlike in top-down analyses, the procedure is analyzed only using a single input, namely ι_{entry}, defined in §5.2.

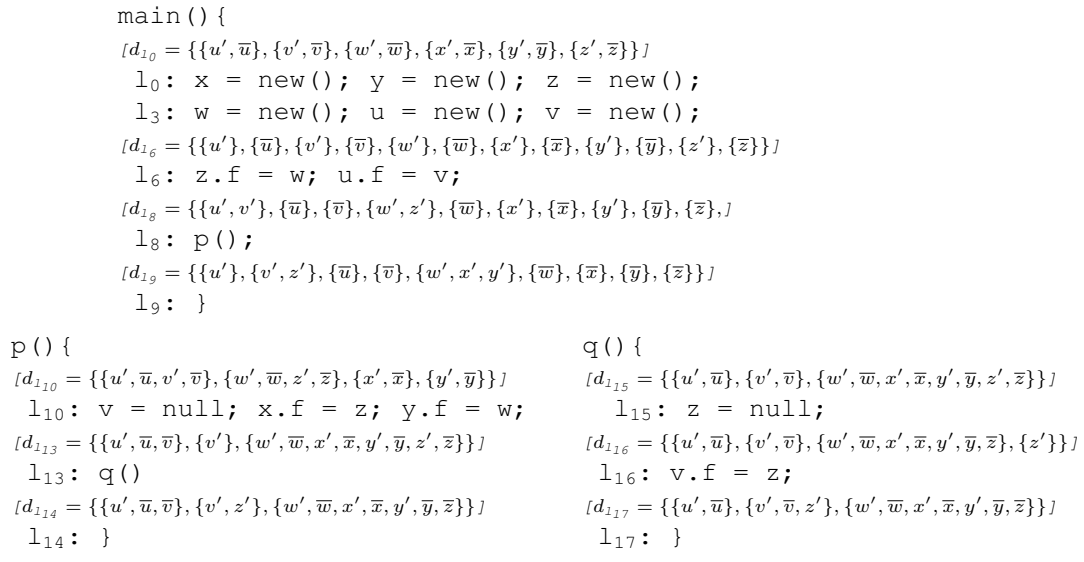


Fig. 3. Example program annotated with abstract states. Abstract state d_{l_i} is computed by the interprocedural *top-down* analysis at program point l_i. All the variables are globals.

5.1 Abstract Domain

The abstract domain $\mathcal{D}$ of the interprocedural connection analyses is obtained by lifting the one used for the *intra*procedural analysis to the *inter*procedural setting. Technically, it is an instantiation of the connection abstract domain $\mathcal{D}(\Upsilon)$ with $\Upsilon = \overline{\mathsf{G}} \cup \mathsf{G}' \cup \dot{\mathsf{G}} \cup \mathsf{L}'$, where $\mathsf{G}' = \{g' \mid g \in \mathsf{G}\}$, $\overline{\mathsf{G}} = \{\overline{g} \mid g \in \mathsf{G}\}$, $\dot{\mathsf{G}} = \{\dot{g} \mid g \in \mathsf{G}\}$, and $\mathsf{L}' = \{x' \mid x \in \mathsf{L}\}$.

The set Υ contains four kinds of elements. Intuitively, the analysis computes at every program point a relation between the objects pointed to by global variables at the entry to the procedure, represented by $\overline{\mathsf{G}}$, and the ones pointed to by global variables and local variables at the current state, represented by G' and L', respectively. As before, abstract states represent partitioning over variables. For technical reasons, described later, Υ also includes the set $\dot{\mathsf{G}}$. The latter is used to compute the effect of procedure calls.

5.2 Interprocedural Top-Down Connection Analysis

The abstract semantics of procedure calls in the top-down analysis is defined in Fig. 4, which we explain below. Intraprocedural commands are handled as described in §4.

When a procedure is entered, local variables of the procedure and all the global variables $\overline{g}$ at the entry to the procedure are initialized to `null`. This is realized by applying the meet operation to d with $R_{\mathsf{G}'}$, effectively, refining the partitioning of d by placing every non-current variable in its own connection set. (We use $d|_S = d \sqcap R_S$ as a shorthand, and say that d is projected on S.) The result, $d|_{\mathsf{G}'}$, represents the connection relation in d between the objects pointed-to by global variables at the call-site. Then, $d|_{\mathsf{G}'}$ is joined with (the particular constant abstract state) ι_{entry}.

The ι_{entry} element abstracts the identity relation between input and output states. It is defined as a partition containing $\{\overline{g}, g'\}$ connection sets for all global variables g.

$[\![p()]\!]^\sharp(d) = [\![\mathsf{return}]\!]^\sharp(([\![C_{\mathsf{body}_p}]\!]^\sharp \circ [\![\mathsf{entry}]\!]^\sharp)(d), d)$, where

$[\![\mathsf{entry}]\!]^\sharp(d) = d|_{\mathsf{G}'} \sqcup \iota_{\mathsf{entry}}$

$[\![\mathsf{return}]\!]^\sharp(d_{\mathsf{exit}}, d_{\mathsf{call}}) = (f_{\mathsf{call}}(d_{\mathsf{call}}) \sqcup f_{\mathsf{exit}}(d_{\mathsf{exit}}|_{\overline{\mathsf{G}}\cup\mathsf{G}'}))|_{\overline{\mathsf{G}}\cup\mathsf{G}'\cup\mathsf{L}}$

$d|_S = d \sqcap R_S \qquad \iota_{\mathsf{entry}} = \bigsqcup_{g\in\mathsf{G}} U_{g'\overline{g}} \qquad R_X = \{\{x \mid x \in X\}\} \cup \{\{x\} \mid x \in \Upsilon \setminus X\}$

$f_{\mathsf{call}}(d) = \{\{\langle n2d(\alpha), n2d(\beta)\rangle \mid \langle\alpha,\beta\rangle \in P\} \mid P \in d\}$, where $n2d(\alpha) = \dot{g}$ if $\alpha = g'$ and α otherwise

$f_{\mathsf{exit}}(d) = \{\{\langle o2d(\alpha), o2d(\beta)\rangle \mid \langle\alpha,\beta\rangle \in P\} \mid P \in d\}$, where $o2d(\alpha) = \dot{g}$ if $\alpha = \overline{g}$ and α otherwise

Fig. 4. The abstract semantics of procedure calls in the top-down analysis. The constant elements $U_{g'\overline{g}}$ are defined in Fig. 2. Note that the renaming functions $f_{\mathsf{call}}(\text{-})$ and $f_{\mathsf{call}}(\text{-})$ are isomorphisms.

Intuitively, the aforementioned join operation records the current value of variable g into $\overline{g}$. Recall that at the entry to a procedure, the "old" value of every global variable is the same as its current value.

$[\![\mathsf{return}]\!]^\sharp$ computes the return value at the caller after the callee returns. It takes two arguments: d_{call}, which represents the partition of variables into connections sets at the call-site, and d_{exit}, which represents the partition at the exit-site of the callee, projected on $\overline{\mathsf{G}} \cup \mathsf{G}'$. This projection emulates the nullification of local variables when exiting a procedure. $[\![\mathsf{return}]\!]^\sharp$ emulates the composition of the input-output relation of the call-site with that of the exit-site using a natural join. The latter is implemented using variables of the form $\dot{g}$: $f_{\mathsf{call}}(d_{\mathsf{call}})$ renames global variables from g' to $\dot{g}$ and $f_{\mathsf{exit}}(d_{\mathsf{exit}})$ renames global variables from $\overline{g}$ to $\dot{g}$. The renamed relations are then joined. Intuitively, the old values $\overline{g}$ of the callee at the exit-site are matched with the current values g' of the caller at the call-site. Finally, the temporary variables are projected away.

Example 3. In Fig. 3, $d_{\mathtt{l}_{13}} = \{\{u', \overline{u}, \overline{v}\}, \{v'\}, \{w', \overline{w}, x', \overline{x}, y', \overline{y}, z', \overline{z}\}\}$ is the abstract state at $\mathtt{l}_{13}$, q()'s call-site in p(), and $d_{\mathtt{l}_{14}}$ is the abstract state at q()'s exit-site.

$[\![q()]\!]^\sharp(d_{\mathtt{l}_{13}}) = [\![\mathsf{return}]\!]^\sharp(([\![C_{\mathsf{body}_q}]\!]^\sharp \circ [\![\mathsf{entry}]\!]^\sharp)(d_{\mathtt{l}_{13}}), d_{\mathtt{l}_{13}})$

$= [\![\mathsf{return}]\!]^\sharp([\![C_{\mathsf{body}_q}]\!]^\sharp(\{\{u', \overline{u}\}, \{v', \overline{v}\}, \{w', \overline{w}, x', \overline{x}, y', \overline{y}, z', \overline{z}\}\}), d_{\mathtt{l}_{13}})$

$= [\![\mathsf{return}]\!]^\sharp(\{\{u', \overline{u}\}, \{v', \overline{v}, z'\}, \{w', \overline{w}, x', \overline{x}, y', \overline{y}, \overline{z}\}\}, \{\{u', \overline{u}, \overline{v}\}, \{v'\}, \{w', \overline{w}, x', \overline{x}, y', \overline{y}, z', \overline{z}\}\})$

$= (f_{\mathsf{exit}}(\{\{u', \overline{u}\}, \{v', \overline{v}, z'\}, \{w', \overline{w}, x', \overline{x}, y', \overline{y}, \overline{z}\}\}|_{\overline{\mathsf{G}}\cup\mathsf{G}'}) \sqcup$
$f_{\mathsf{call}}(\{\{u', \overline{u}, \overline{v}\}, \{v'\}, \{w', \overline{w}, x', \overline{x}, y', \overline{y}, z', \overline{z}\}\}))|_{\overline{\mathsf{G}}\cup\mathsf{G}'\cup\mathsf{L}}$

$= (\{\{u', \dot{u}\}, \{v', \dot{v}, z'\}, \{w', \dot{w}, x', \dot{x}, y', \dot{y}, \dot{z}\}\} \sqcup \{\dot{u}, \overline{u}, \overline{v}\}, \{\dot{v}\}, \{\dot{w}, \overline{w}, \dot{x}, \overline{x}, \dot{y}, \overline{y}, \dot{z}, \overline{z}\}\})|_{\overline{\mathsf{G}}\cup\mathsf{G}'\cup\mathsf{L}}$

$= \{\{u', \dot{u}, \overline{u}, \overline{v}\}, \{v', \dot{v}, z'\}, \{w', \dot{w}, \overline{w}, x', \dot{x}, \overline{x}, y', \dot{y}, \overline{y}, \dot{z}, \overline{z}\}\}|_{\overline{\mathsf{G}}\cup\mathsf{G}'\cup\mathsf{L}}$

$= \{\{u', \overline{u}, \overline{v}\}, \{v', z'\}, \{w', \overline{w}, x', \overline{x}, y', \overline{y}, \overline{z}\}\} = d_{\mathtt{l}_{14}}$

Lemma 3 (Soundness of the Top-Down Analysis). *The abstract semantics of the top-down interprocedural connection analysis is an over-approximation of the standard concrete semantics for heap manipulating programs [4].*

The crux of the proof is the observation that the abstract transfer of the return statements is sound because the "old" values of the global variables of a procedure are never modified and are the same as their "current" values when it was invoked.

5.3 Bottom Up Compositional Connection Analysis

The abstract semantics $[\![-]\!]^\sharp_{\mathsf{BU}}(-)$ of procedure calls in the bottom-up analysis is defined in Eq. 1. Again, intraprocedural commands are handled as described in §4.

$$[\![p()]\!]^\sharp_{\mathsf{BU}}(d) = [\![\mathsf{return}]\!]^\sharp([\![C_{\mathsf{body}_p}]\!]^\sharp(\iota_{\mathsf{entry}}), d) \tag{1}$$

$[\![p()]\!]^\sharp_{\mathsf{BU}}(d)$ and $[\![p()]\!]^\sharp(d)$, defined in Fig. 4, differ in the way in which the value of the first argument to $[\![\mathsf{return}]\!]^\sharp(\cdot, d)$ is computed: $[\![p()]\!]^\sharp_{\mathsf{BU}}(d)$ uses the abstract state resulting at the exit of p's body when it is analyzed with state ι_{entry}. Hence, it uses the same value at every call. In contrast, $[\![p()]\!]^\sharp(d)$ computes that argument by analyzing the call to $\mathtt{p}()$ with the particular *call state* d. Note that as a corollary of the theorem we get that the bottom-up interprocedural connection analysis is sound.

Theorem 1 (Coincidence). $\forall C \in \mathit{Commands}. \forall d \in \mathcal{D}. [\![C]\!]^\sharp_{\mathsf{BU}}(d) = [\![C]\!]^\sharp(d)$.

In the rest of the section, we sketch the main arguments in the proof of The. 1, in lieu of more formal mathematical arguments, which are shown in the proof of [4, The. 22]. We focus on the case where C is a procedure invocation, i.e., $C = p()$.

Notation. We denote by $\mathcal{D}_X$ the sublattice representing the closed interval $[\bot, R_X]$ for a set $\emptyset \neq X \subseteq \Upsilon$, which consists of all the elements between $\bot$ and R_X. For example, $\mathcal{D}_{\overline{\mathsf{G}}}$ includes only partitions where all the variables not in $\overline{\mathsf{G}}$ are in singleton sets. We define the sets $S_{\widehat{x}}$ and $U_{\widehat{x}\widehat{y}}$, where $\widehat{x}$ is either x', $\overline{x}$, or $\dot{x}$ in the same way as $S_{x'}$ and $U_{x'y'}$ are defined in Fig. 2. For example, $U_{\overline{x}y'} = \{\{\overline{x}, y'\}\} \cup \{\{\alpha\} \mid \alpha \in \Upsilon \setminus \{\overline{x}, y'\}\}$.

5.3.1 Uniform Representation of Entry Abstract States The abstract states at the entry to procedures in the top-down analysis are *uniform*: for every global variable g, we have a connection set containing only $\overline{g}$ and g'. This is a result of the definition of function entry, which projects abstract call states on G' and then joins the result with the ι_{entry} element. The projection results in an abstract state where all connection sets containing more than a single element are comprised only of primed variables. Then, after joining $d|_{\mathsf{G}'}$ with ι_{entry}, each old variable $\overline{g}$ resides in the same partition as its corresponding current primed variable g'. For example, see $d_{1_{10}}$ in Fig. 3.

We point out that the uniformity of the entry states is due to the property of ι_{entry} that its connection sets are comprised of pairs of variables of the form $\{x', \overline{x}\}$. One important implication of this uniformity is that every entry abstract state d to any procedure has a dual representation. In one representation, d is the join of ι_{entry} with some elements $U_{x'y'} \in \mathcal{D}_{\mathsf{G}'}$. In the other representation, d is expressed as the join of ι_{entry} with some elements $U_{\overline{xy}} \in \mathcal{D}_{\overline{\mathsf{G}}}$. In the following, we use the function o that replaces relationships among current variables by those among old ones: $o(U_{x'y'}) = U_{\overline{xy}}$; and $o(d)$ is the least upper bounds of ι_{entry} and elements $U_{\overline{xy}}$ for all x, y such that x' and y' are in the same connection set of d.

Example 4. $d_{1_{10}}$ is the abstract element at the entry point of procedure p of Fig. 3.
$\iota_{\mathsf{entry}} \sqcup (U_{u'v'} \sqcup U_{w'z'}) = o(\iota_{\mathsf{entry}} \sqcup (U_{u'v'} \sqcup U_{w'z'})) = \iota_{\mathsf{entry}} \sqcup o(U_{u'v'} \sqcup U_{w'z'})$
$= \iota_{\mathsf{entry}} \sqcup (U_{\overline{uv}} \sqcup U_{\overline{wz}}) = \{\{u', \overline{u}, v', \overline{v}\}, \{w', \overline{w}, z', \overline{z}\}, \{x', \overline{x}\}, \{y', \overline{y}\}\} = d_{1_{10}}$.

Delayed Evaluation of the Effect of Calling Contexts Elements of the form $U_{\overline{xy}}$, coming from $\mathcal{D}_{\overline{\mathsf{G}}}$, are smaller than or equal to the meet elements $S_{x'}$ of intraprocedural statements. This is because for any $x, y \in \mathsf{G}$ it holds that

$$S_{x'} = \{\{x'\}\} \cup \{\{z | z \in \Upsilon \setminus \{x'\}\}\} \sqsupseteq \{\{\overline{x}, \overline{y}\}\} \cup \{\{z\} | z \in \Upsilon \setminus \{\overline{x}, \overline{y}\}\} = U_{\overline{xy}} .$$

In Lem. 1 of §4 we proved that the semantics of the connection analysis is conditionally adaptable. Thus, computing the composed effect of any sequence τ of intraprocedural transformers on an entry state of the form $d_0 \sqcup U_{\overline{x}_1\overline{y}_1} \ldots \sqcup U_{\overline{x}_n\overline{y}_n}$ results in an element of the form $d'_0 \sqcup U_{\overline{x}_1\overline{y}_1} \ldots \sqcup U_{\overline{x}_n\overline{y}_n}$, where d'_0 results from applying the transformers in

τ on d_0. Using the observations made in §5.3.1, this means that for any abstract element d resulting at a call-site there exists an element $d_2 \in \mathcal{D}_{\overline{\mathsf{G}}}$ which is a join of elements of the form $U_{\overline{xy}} \in \mathcal{D}_{\overline{\mathsf{G}}}$, such that $d = d_1 \sqcup d_2$, and $d_1 = [\![\tau]\!]^\sharp(\iota_{\mathsf{entry}})$.

$$d = d_1 \sqcup U_{\overline{x}_1\overline{y}_1} \ldots \sqcup U_{\overline{x}_n\overline{y}_n} \,. \tag{2}$$

Example 5. The abstract state at entry point of p is $d_{1_{10}} = \iota_{\mathsf{entry}} \sqcup (U_{\overline{uv}} \sqcup U_{\overline{wz}})$. (See Exa. 4.) The sequence of commands at 1_{10} is $C := \mathtt{v} = \mathtt{null}; \mathtt{x.f} = \mathtt{z}; \mathtt{y.f} = \mathtt{w}$. Thus, $d_{1_{13}} = [\![C]\!]^\sharp(d_{1_{10}})$. Note that $d_{1_{13}}$ can also be computed using the effect of C to ι_{entry}:

$$\begin{aligned}
[\![C]\!]^\sharp(\iota_{\mathsf{entry}}) \sqcup (U_{\overline{wz}} \sqcup U_{\overline{uv}}) &= \{\{u', \overline{u}\}, \{v'\}, \{\overline{v}\}, \{w', \overline{w}, y', \overline{y}\}, \{x', \overline{x}, z', \overline{z}\}\} \sqcup (U_{\overline{uv}} \sqcup U_{\overline{wz}}) \\
&= \{\{u', \overline{u}, \overline{v}\}, \{v'\}, \{w', \overline{w}, x', \overline{x}, y', \overline{y}, z', \overline{z}\}\} = d_{1_{13}} \\
[\![C]\!]^\sharp(\iota_{\mathsf{entry}}) &= [\![\mathtt{v} = \mathtt{null}; \mathtt{x.f} = \mathtt{z}; \mathtt{y.f} = \mathtt{w}]\!]^\sharp(\{\{u', \overline{u}\}, \{v', \overline{v}\}, \{w', \overline{w}\}, \{x', \overline{x}\}, \{y', \overline{y}\}, \{z', \overline{z}\}\}) \\
&= \{\{u', \overline{u}\}, \{v'\}, \{\overline{v}\}, \{w', \overline{w}, y', \overline{y}\}, \{x', \overline{x}, z', \overline{z}\}\}
\end{aligned}$$

5.3.2 Counterpart Representation for Calling Contexts The previous reasoning ensures that any abstract value at the call-site to a procedure $p()$ is of the form $d_1 \sqcup d_2$, where $d_2 \in \mathcal{D}_{\overline{\mathsf{G}}}$ and, thus, is a join of elements of form $U_{\overline{xy}}$. Furthermore, the state resulting at the entry of $p()$ when the calling context is $d_1 \sqcup U_{\overline{xy}}$ can be obtained either directly from d_1 or after merging two of d_1's connection sets. Note that the need to merge occurs only if there are variables w' and z' such that w' and $\overline{x}$ are in one of the connection sets and z' and $\overline{y}$ are in another. This means that the effect of $U_{\overline{xy}}$ on the entry state can be expressed via primed variables: $d_1 \sqcup U_{\overline{xy}} = d_1 \sqcup U_{w'z'}$. Thus, if the abstract state at the call-site is $d_1 \sqcup d_2$, then there is an element $d_2' \in \mathcal{D}_{\mathsf{G}'}$ such that

$$(d_1 \sqcup d_2)|_{\mathsf{G}'} = d_1|_{\mathsf{G}'} \sqcup d_2' \tag{3}$$

We refer to the element $d_2' \in \mathcal{D}_{\mathsf{G}'}$, which can be used to represent the effect of $d_2 \in \mathcal{D}_{\overline{\mathsf{G}}}$ at the call-site as d_2's *counterpart*, and denote it by $\widehat{d_2}$.

Example 6. Let $d_1 = \{\{u', \overline{u}\}, \{v'\}, \{\overline{v}\}, \{w', \overline{w}, y', \overline{y}\}, \{x', \overline{x}, z', \overline{z}\}\}$ and $d_2 = U_{\overline{wz}}$. Joining d_1 with $U_{\overline{wz}}$ causes connection sets $[\overline{w}]$ and $[\overline{z}]$ to be merged, and, consequently, $[y']$ and $[x']$ are merged, since $[y'] = [\overline{w}]$ and $[x'] = [\overline{z}]$. Therefore, for $\widehat{d_2} = U_{x'z'}$ it holds that $(d_1 \sqcup d_2)|_{\mathsf{G}'} = \{\{u', \overline{u}\}, \{v'\}, \{\overline{v}\}, \{w', \overline{w}, y', \overline{y}, x', \overline{x}, z', \overline{z}\}\}|_{\mathsf{G}'} = \{\{u'\}, \{v'\}, \{w', y', x', z'\}\}$. Similarly, $d_1|_{\mathsf{G}'} \sqcup \widehat{d_2} = \{\{u'\}, \{v'\}, \{w', y'\}, \{x', z'\}\} \sqcup U_{x'z'} = \{\{u'\}, \{v'\}, \{w', y', x', z'\}\}$.

Representing Entry States with Counterparts The above facts imply that we can represent an abstract state d at the call-site as $d = d_1 \sqcup d_2$, where $d_2 = d_3 \sqcup d_4$ for some $d_3, d_4 \in \mathcal{D}_{\overline{\mathsf{G}}}$ such that: (i) d_3 is a join of the elements of the form $U_{\overline{xy}}$ such that $\overline{x}$ and $\overline{y}$ reside in d_1 in different partitions, which also contain current (primed) variables, and thus possibly affect the entry state, and (ii) d_4 is a join of all the other elements $U_{\overline{xy}} \in \mathcal{D}_{\overline{\mathsf{G}}}$, which are needed to represent d in this form, but either $\bar{x}$ or $\bar{y}$ resides in the same partition in d_1 or one of them is in a partition containing only old variables. Recall that there is an element $d_3' = \widehat{d_3}$ that joins elements of the form $U_{x'y'}$ such that $d_1 \sqcup d_3 = d_1 \sqcup d_3'$, and therefore

$$d = d_1 \sqcup d_3 \sqcup d_4 = d_1 \sqcup d_3' \sqcup d_4 \,. \tag{4}$$

Thus, after applying the entry's semantics, we get that abstract states at the entry point of procedure are always of the form

$$[\![\mathsf{entry}]\!]^\sharp(d) = (d_1 \sqcup d_3' \sqcup d_4)|_{\mathsf{G}'} \sqcup \iota_{\mathsf{entry}} = (d_1 \sqcup d_3')|_{\mathsf{G}'} \sqcup \iota_{\mathsf{entry}} = (d_1|_{\mathsf{G}'} \sqcup d_3') \sqcup \iota_{\mathsf{entry}}$$

where d_3' represents the effect of $d_3 \sqcup d_4$ on partitions containing current variables g' in d_1. The second equality holds because the *modularity of* $R_{G'}$: d_3' joins elements of form

$U_{x'y'}$ and $U_{x'y'} \sqsubseteq R_{G'}$. This implies that every state d_0 at an entry point to a procedure is of the following form:

$$\begin{aligned} d_0 &= \iota_{\mathsf{entry}} \sqcup \overbrace{(U_{x'y'} \ldots \sqcup U_{x'_l y'_l})}^{d_1|_{\mathsf{G}'}} \sqcup \overbrace{(U_{x'_{l+1} y'_{l+1}} \ldots \sqcup U_{x'_n y'_n})}^{d'_3} \\ &= \iota_{\mathsf{entry}} \sqcup o(U_{x'_1 y'_1} \ldots \sqcup U_{x'_n y'_n}) = \iota_{\mathsf{entry}} \sqcup U_{\overline{x}_1 \overline{y}_1} \sqcup \ldots \sqcup U_{\overline{x}_n \overline{y}_n} \end{aligned} \tag{5}$$

The second equality is obtained using the dual representation of entry state (see §5.3.1) and the third one is justified because $o(\text{-})$ is an isomorphism.

Example 7. The abstract state at the call-site of procedure $\mathsf{q}()$ is $d_{1_{13}} = d_1 \sqcup d_2$ where $d_1 = \{\{u', \overline{u}\}, \{v'\}, \{\overline{v}\}, \{w', \overline{w}, y', \overline{y}\}, \{x', \overline{x}, z', \overline{z}\}\}$ and $d_2 = U_{\overline{uv}} \sqcup U_{\overline{wz}}$. (See the first equality in Exa. 5.) In Exa. 6 we showed that $U_{\overline{wz}}$ affects the relations in d_1 between current variables and that $\widehat{U_{\overline{wz}}} = U_{x'y'}$. In contrast, joining the result with $U_{\overline{uv}}$ has no effect on relations between current variables, because the connection set $[\overline{v}]$ does not contain any current variable. Indeed, $d_1 \sqcup U_{\overline{wz}} \sqcup U_{\overline{uv}} = d_1 \sqcup U_{x'y'} \sqcup U_{\overline{uv}}$. Following the reasoning above, consider the abstract state at the entry to procedure $\mathsf{q}()$

$$\begin{aligned} d_{1_{15}} &= \{\{u', \overline{u}\}, \{v', \overline{v}\}, \{w', \overline{w}, x', \overline{x}, y', \overline{y}, z', \overline{z}\}\} \\ &= \iota_{\mathsf{entry}} \sqcup \{\{u'\}, \{\overline{u}\}, \{v'\}, \{\overline{v}\}, \{w', y'\}, \{\overline{w}\}, \{x', z'\}, \{\overline{x}\}, \{\overline{y}\}, \{\overline{z}\}\} \sqcup U_{\overline{wz}} \\ &= \iota_{\mathsf{entry}} \sqcup d_1|_{\mathsf{G}'} \sqcup U_{x'y'} \end{aligned}$$

Adapting the Result for Different Contexts We now show that the interprocedural connection analysis can be done compositionally. Intuitively, we show that the effect of the caller's calling context can be carried over procedure invocations. Alternatively, the effect of the callee on the caller's context can be adapted unconditionally for different caller's calling contexts. The proof goes by induction on the structure of the program. We sketch the proof for the case where $C = p()$.

In Eq. 4 we showed that every abstract value that arises at the call-site is of the form $d_1 \sqcup d_3 \sqcup d_4$, where $d_3, d_4 \in \mathcal{D}_{\overline{\mathsf{G}}}$. Thus, we need to show for any $d_1 \neq \bot$ that

$$[\![p()]\!]^\sharp(d_1 \sqcup d_3 \sqcup d_4) = [\![p()]\!]^\sharp(d_1) \sqcup d_3 \sqcup d_4\,. \tag{6}$$

According to the *top-down* abstract semantics the effect of invoking $p()$ is

$$[\![p()]\!]^\sharp(d) = [\![\mathsf{return}]\!]^\sharp(d_{\mathsf{exit}}, d) = [\![\mathsf{return}]\!]^\sharp\big(((([\![C_{\mathsf{body}_p}]\!]^\sharp \circ [\![\mathsf{entry}]\!]^\sharp)(d)\big), d\big)\,.$$

Because d is of the form $d_1 \sqcup d_3 \sqcup d_4$, we can write d_{exit} as below, where first equalities are mere substitutions based on observations we made before and the last one comes from the induction assumption.

$$\begin{aligned} d_{\mathsf{exit}} &= [\![C_{\mathsf{body}_p}]\!]^\sharp([\![\mathsf{entry}]\!]^\sharp(d_1 \sqcup d_3 \sqcup d_4)) = [\![C_{\mathsf{body}_p}]\!]^\sharp(((d_1 \sqcup d_3 \sqcup d_4)|_{\mathsf{G}'}) \sqcup \iota_{\mathsf{entry}}) \\ &= [\![C_{\mathsf{body}_p}]\!]^\sharp(((d_1 \sqcup d_3)|_{\mathsf{G}'}) \sqcup \iota_{\mathsf{entry}}) = [\![C_{\mathsf{body}_p}]\!]^\sharp(d_1|_{\mathsf{G}'} \sqcup d'_3 \sqcup \iota_{\mathsf{entry}}) \\ &= [\![C_{\mathsf{body}_p}]\!]^\sharp(d_1|_{\mathsf{G}'} \sqcup o(d'_3) \sqcup \iota_{\mathsf{entry}}) = [\![C_{\mathsf{body}_p}]\!]^\sharp(d_1|_{\mathsf{G}'} \sqcup \iota_{\mathsf{entry}}) \sqcup o(d'_3) \end{aligned} \tag{7}$$

When applying the return semantics, we first compute the natural join and then remove the temporary variables. Therefore, we get

$$[\![p()]\!]^\sharp(d) = (f_{\mathsf{call}}(d_1 \sqcup d_3 \sqcup d_4) \sqcup f_{\mathsf{exit}}([\![C_{\mathsf{body}_p}]\!]^\sharp(d_1|_{\mathsf{G}'} \sqcup \iota_{\mathsf{entry}}) \sqcup o(d'_3)))|_{\overline{\mathsf{G}} \cup \mathsf{G}' \cup \mathsf{L}}\,.$$

Eq. 8 shows the result of computing the inner parentheses. The first equality is by the definition of d'_3 and the last equality is by the isomorphism of $f_{\mathsf{call}}(\text{-})$ and $f_{\mathsf{exit}}(\text{-})$.

$$\begin{aligned} & f_{\mathsf{call}}(d_1 \sqcup d_3 \sqcup d_4) \sqcup f_{\mathsf{exit}}([\![C_{\mathsf{body}_p}]\!]^\sharp(d_1|_{\mathsf{G}'} \sqcup \iota_{\mathsf{entry}}) \sqcup o(d'_3)) \\ &= f_{\mathsf{call}}(d_1 \sqcup d'_3 \sqcup d_4) \sqcup f_{\mathsf{exit}}([\![C_{\mathsf{body}_p}]\!]^\sharp(d_1|_{\mathsf{G}'} \sqcup \iota_{\mathsf{entry}}) \sqcup o(d'_3)) \\ &= f_{\mathsf{call}}(d'_3) \sqcup f_{\mathsf{call}}(d_1 \sqcup d_4) \sqcup f_{\mathsf{exit}}(o(d'_3)) \sqcup f_{\mathsf{exit}}([\![C_{\mathsf{body}_p}]\!]^\sharp(d_1|_{\mathsf{G}'} \sqcup \iota_{\mathsf{entry}})) \end{aligned} \tag{8}$$

Note, among the join arguments, $f_{\mathsf{exit}}(o(d'_3))$ and $f_{\mathsf{call}}(d'_3)$. Let's look at the first element. $o(d'_3)$ replaces all the occurrences of $U_{x'y'}$ in d'_3 with $U_{\overline{xy}}$. f_{exit} replaces all the occurrences of $U_{\overline{xy}}$ in $o(d'_3)$ with $U_{\dot{x}\dot{y}}$. Thus, the first element is $U_{\dot{x}_1\dot{y}_1} \sqcup \ldots \sqcup U_{\dot{x}_n\dot{y}_n}$ which is the result of replacing in d'_3 all the occurrences of $U_{x'y'}$ with $U_{\dot{x}\dot{y}}$. Let's look now at the second element. f_{call} replaces all occurrences of $U_{x'y'}$ in d'_3 with $U_{\dot{x}\dot{y}}$. Thus, also the second element is $U_{\dot{x}_1\dot{y}_1} \sqcup \ldots \sqcup U_{\dot{x}_n\dot{y}_n}$, i.e, $f_{\mathsf{call}}(d'_3) = f_{\mathsf{exit}}(o(d'_3))$, and we get

$$\begin{aligned}(8) &= f_{\mathsf{call}}(d'_3) \sqcup f_{\mathsf{call}}(d_1 \sqcup d_4) \sqcup f_{\mathsf{exit}}([\![C_{\mathsf{body}_p}]\!]^\sharp(d_1|_{\mathsf{G}'} \sqcup \iota_{\mathsf{entry}}))\\ &= f_{\mathsf{call}}(d_3 \sqcup d_1 \sqcup d_4) \sqcup f_{\mathsf{exit}}([\![C_{\mathsf{body}_p}]\!]^\sharp(d_1|_{\mathsf{G}'} \sqcup \iota_{\mathsf{entry}}))\\ &= f_{\mathsf{call}}(d_1) \sqcup f_{\mathsf{exit}}([\![C_{\mathsf{body}_p}]\!]^\sharp(d_1|_{\mathsf{G}'} \sqcup \iota_{\mathsf{entry}}) \sqcup (d_3 \sqcup d_4) = [\![p()]\!]^\sharp(d_1) \sqcup d_3 \sqcup d_4\end{aligned}$$

The first equality is by the idempotence of $\sqcup$. The second equality is by the isomorphism of f_{call} and Eq. 4. To justify the third equality, recall (Eq. 2) that d_3 and d_4 are both of form $U_{\overline{x}_1\overline{y}_1} \sqcup \ldots \sqcup U_{\overline{x}_n\overline{y}_n}$ and that $f_{\mathsf{call}}(d)$ only replaces g' occurrences in d; and thus $f_{\mathsf{call}}(U_{\overline{x}_1\overline{y}_1} \sqcup \ldots \sqcup U_{\overline{x}_n\overline{y}_n}) = U_{\overline{x}_1\overline{y}_1} \sqcup \ldots \sqcup U_{\overline{x}_n\overline{y}_n}$.

Example 8. By Exa. 5, $d_{1_{13}} = \{\{u', \overline{u}\}, \{v'\}, \{\overline{v}\}, \{w', \overline{w}, y', \overline{y}\}, \{x', \overline{x}, z', \overline{z}\}\} \sqcup (U_{\overline{uv}} \sqcup U_{\overline{wz}})$. Let $d_1 = \{\{u', \overline{u}\}, \{v'\}, \{\overline{v}\}, \{w', \overline{w}, y', \overline{y}\}, \{x', \overline{x}, z', \overline{z}\}\}$ and $d_2 = U_{\overline{uv}} \sqcup U_{\overline{wz}}$. Thus, $d_{1_{13}} = d_1 \sqcup d_2$ and $d_2 \in \mathcal{D}_{\overline{\mathsf{G}}}$. Let's compute (i) $[\![q()]\!]^\sharp(d_1)$ and (ii) $[\![q()]\!]^\sharp(d_1) \sqcup d_2$.

$$\begin{aligned}&[\![q()]\!]^\sharp(d_1) = [\![\mathsf{return}]\!]^\sharp(([\![C_{\mathsf{body}_q}]\!]^\sharp \circ [\![\mathsf{entry}]\!]^\sharp)(d_1), d_1)\\ &= [\![\mathsf{return}]\!]^\sharp(([\![C_{\mathsf{body}_q}]\!]^\sharp \circ [\![\mathsf{entry}]\!]^\sharp)(\{\{u', \overline{u}\}, \{v'\}, \{\overline{v}\}, \{w', \overline{w}, y', \overline{y}\}, \{x', \overline{x}, z', \overline{z}\}\}), d_1)\\ &= [\![\mathsf{return}]\!]^\sharp([\![C_{\mathsf{body}_q}]\!]^\sharp(\{\{u', \overline{u}\}, \{v', \overline{v}\}, \{w', \overline{w}, y', \overline{y}\}, \{x', \overline{x}, z', \overline{z}\}\}), d_1)\\ &= [\![\mathsf{return}]\!]^\sharp(\{\{u', \overline{u}\}, \{v', \overline{v}, z'\}, \{w', \overline{w}, y', \overline{y}\}, \{x', \overline{x}, \overline{z}\}\}, \{\{u', \overline{u}\}, \{v'\}, \{\overline{v}\}, \{w', \overline{w}, y', \overline{y}\}, \{x', \overline{x}, z', \overline{z}\}\})\\ &= (f_{\mathsf{exit}}(\{\{u', \overline{u}\}, \{v', \overline{v}, z'\}, \{w', \overline{w}, y', \overline{y}\}, \{x', \overline{x}, \overline{z}\}\}|_{\overline{\mathsf{G}} \cup \mathsf{G}'}) \sqcup\\ &\qquad f_{\mathsf{call}}(\{\{u', \overline{u}\}, \{v'\}, \{\overline{v}\}, \{w', \overline{w}, y', \overline{y}\}, \{x', \overline{x}, z', \overline{z}\}\}))|_{\overline{\mathsf{G}} \cup \mathsf{G}' \cup \mathsf{L}}\\ &= (\{\{u', \dot{u}\}, \{v', \dot{v}, z'\}, \{w', \dot{w}, y', \dot{y}\}, \{x', \dot{x}, \dot{z}\}\} \sqcup \{\{\dot{u}, \overline{u}\}, \{\dot{v}\}, \{\overline{v}\}, \{\dot{w}, \overline{w}, \dot{y}, \overline{y}\}, \{\dot{x}, \overline{x}, \dot{z}, \overline{z}\}\})_{\overline{\mathsf{G}} \cup \mathsf{G}' \cup \mathsf{L}}\\ &= (\{\{u', \dot{u}, \overline{u}\}, \{v', \dot{v}, z'\}, \{\overline{v}\}, \{w', \dot{w}, \overline{w}, y', \dot{y}, \overline{y}\}, \{x', \dot{x}, \overline{x}, \dot{z}, \overline{z}\}\})|_{\overline{\mathsf{G}} \cup \mathsf{G}' \cup \mathsf{L}}\\ &= \{\{u', \overline{u}\}, \{v', z'\}, \{\overline{v}\}, \{w', \overline{w}, y', \overline{y}\}, \{x', \overline{x}, \overline{z}\}\}\\ &[\![q()]\!]^\sharp(d_1) \sqcup d_2 = \{\{u', \overline{u}\}, \{v', z'\}, \{\overline{v}\}\}, \{w', \overline{w}, y', \overline{y}\}, \{x', \overline{x}, \overline{z}\}\} \sqcup (U_{\overline{uv}} \sqcup U_{\overline{wz}})\\ &\qquad = \{\{u', \overline{u}, \overline{v}\}, \{z', v'\}, \{w', \overline{w}, x', \overline{x}, y', \overline{y}, \overline{z}\}\} d_{1_{14}} = [\![q()]\!]^\sharp(d_{1_{13}}) = [\![q()]\!]^\sharp(d_1 \sqcup d_2)\end{aligned}$$

Precision Coincidence We combine the observations we made to informally show the coincidence result between the top-down and the bottom-up semantics (The. 1). By Eq. 4, every state d at a call-site can be represented as $d = d_1 \sqcup d_3 \sqcup d_4$, where $d_3, d_4 \in \mathcal{D}_{\overline{\mathsf{G}}}$. Furthermore, there exists $d'_3 = \widehat{d_3} \in \mathcal{D}_{\mathsf{G}'}$ such that $d_1 \sqcup d_3 \sqcup d_4 = d_1 \sqcup d'_3 \sqcup d_4$. We also showed that for every command C and every $d = d_1 \sqcup d_3 \sqcup d_4$, such that $d_3, d_4 \in \mathcal{D}_{\overline{\mathsf{G}}}$, it holds that $[\![C]\!]^\sharp(d_1 \sqcup d_3 \sqcup d_4) = [\![C]\!]^\sharp(d_1) \sqcup d_3 \sqcup d_4$. Finally,

$$\begin{aligned}[\![p()]\!]^\sharp(d) &= [\![\mathsf{return}]\!]^\sharp([\![C_{\mathsf{body}_p}]\!]^\sharp([\![\mathsf{entry}]\!]^\sharp(d)), d)\\ &= [\![\mathsf{return}]\!]^\sharp([\![C_{\mathsf{body}_p}]\!]^\sharp(d_1|_{\mathsf{G}'} \sqcup \iota_{\mathsf{entry}} \sqcup d'_3), d)\\ &= [\![\mathsf{return}]\!]^\sharp([\![C_{\mathsf{body}_p}]\!]^\sharp(\iota_{\mathsf{entry}} \sqcup o(d_1|_{\mathsf{G}'}) \sqcup o(d'_3)), d)\\ &= [\![\mathsf{return}]\!]^\sharp([\![C_{\mathsf{body}_p}]\!]^\sharp(\iota_{\mathsf{entry}}) \sqcup o(d_1|_{\mathsf{G}'}) \sqcup o(d'_3), d_1 \sqcup d_3 \sqcup d_4)\\ &= [\![\mathsf{return}]\!]^\sharp([\![C_{\mathsf{body}_p}]\!]^\sharp(\iota_{\mathsf{entry}}), d_1 \sqcup d_3 \sqcup d_4) = [\![p()]\!]^\sharp_{\mathsf{BU}}(d)\ .\end{aligned}$$

The second equality is by Eq. 7. The third equality holds because $d'_3, d_1|_{\mathsf{G}'} \in \mathcal{D}_{\mathsf{G}'}$ and by Eq. 5. The forth equality holds since $o(d'_3), o(d_1|_{\mathsf{G}'}) \in \mathcal{D}_{\overline{\mathsf{G}}}$ and by Eq. 8. The fifth equality holds because we can remove $o(d'_3)$ as $f_{\mathsf{exit}}(o(d'_3))$ is redundant in the natural join. Using a similar reasoning, we can remove $f_{\mathsf{exit}}(o(d_1|_{\mathsf{G}'}))$, since f_{exit} is an isomorphism and $f_{\mathsf{exit}}(o(d_1|_{\mathsf{G}'})) = f_{\mathsf{call}}(d_1|_{\mathsf{G}'}) \sqsubseteq f_{\mathsf{call}}(d_1)$.

Example 9. Let's compute the result of applying $\mathtt{p}()$ to d_{1_8} using the bottom-up semantics, starting by computing $[\![C_{\mathsf{body}_p}]\!]^\sharp(\iota_{\mathsf{entry}})$ and then $[\![p()]\!]^\sharp_{\mathsf{BU}}(d_{1_8})$.

$$
\begin{aligned}
&[\![C_{\mathsf{body}_p}]\!]^\sharp(\iota_{\mathsf{entry}}) = \{\{u',\overline{u}\},\{\overline{v}\},\{v',z'\},\{w',\overline{w},y',\overline{y}\},\{x',\overline{x},\overline{z}\}\}\\
&[\![p()]\!]^\sharp_{\mathsf{BU}}(d_{1_8}) = [\![\mathsf{return}]\!]^\sharp([\![C_{\mathsf{body}_p}]\!]^\sharp(\iota_{\mathsf{entry}}), d_{1_8})\\
&= (f_{\mathsf{exit}}(\{\{u',\overline{u}\},\{v',z'\},\{\overline{v}\},\{w',\overline{w},y',\overline{y}\},\{x',\overline{x},\overline{z}\}\})\\
&\quad \sqcup f_{\mathsf{call}}(\{\{u',v'\},\{\overline{u}\},\{\overline{v}\},\{\overline{w}\},\{x'\},\{\overline{x}\},\{y'\},\{\overline{y}\},\{w',z'\},\{\overline{z}\}\}|_{\overline{\mathsf{G}}\cup\mathsf{G}'}))|_{\overline{\mathsf{G}}\cup\mathsf{G}'\cup\mathsf{L}}\\
&= (\{\{u',\dot{u}\},\{v',z'\},\{\dot{v}\},\{w',\dot{w},y',\dot{y}\},\{x',\dot{x},\dot{z}\}\}\\
&\quad \sqcup\{\{\dot{u},\dot{v}\},\{\overline{u}\},\{\overline{v}\},\{\overline{w}\},\{\dot{x}\},\{\overline{x}\},\{\dot{y}\},\{\overline{y}\},\{\dot{w},\dot{z}\},\{\overline{z}\}\})|_{\overline{\mathsf{G}}\cup\mathsf{G}'\cup\mathsf{L}}\\
&= (\{\{u',\dot{u},\dot{v}\},\{v',z'\},\{\overline{u}\},\{\overline{v}\},\{w',\dot{w},x',\dot{x},y',\dot{y},\dot{z}\},\{\overline{w}\},\{\overline{x}\},\{\overline{y}\},\{\overline{z}\}\})|_{\overline{\mathsf{G}}\cup\mathsf{G}'\cup\mathsf{L}}\\
&= \{\{u'\},\{v',z'\},\{\overline{u}\},\{\overline{v}\},\{w',x',y'\},\{\overline{w}\},\{\overline{x}\},\{\overline{y}\},\{\overline{z}\}\} = d_{1_9} = [\![p()]\!]^\sharp(d_{1_8})
\end{aligned}
$$

6 Implementation and Experimental Evaluation

We implemented three versions of the connection analysis: the original top-down version [14], our modified top-down version, and our modular bottom-up version that coincides in precision with the modified top-down version. We next describe these versions.

The abstract transformer of the destructive update statements $x.f = y$ in [14] does not satisfy the requirements described in §4; its effect depends on the abstract state. Specifically, the connection sets of x and y are not merged if x or y points to `null` in all the executions leading to this statement. We therefore conservatively modified the analysis to satisfy our requirements, by changing the abstract transformer to always merge x's and y's connection sets. Our bottom-up modular analysis that coincides with this modified top-down analysis operates in two phases. The first phase computes a summary for every procedure by analyzing it with an input state ι_{entry}. The summary over-approximates relations between all possible inputs of this procedure and each program point in the body of the procedure. The second phase is a chaotic iteration algorithm which propagates values from callers to callees using the precomputed summaries, and is similar to the second phase of the interprocedural functional algorithm of [28, Fig. 7].

We implemented the aforementioned versions of connection analysis using Chord [26] and applied them to the five Java benchmark programs listed in Tab. 1. (For space reasons, however, we do not discuss the modified top-down version of connection analysis.) They include two programs (grande2 and grande3) from the Java Grande benchmark suite and two (antlr and bloat) from the DaCapo benchmark suite. We excluded programs from these suites that use multi-threading, since our analyses assume sequential programs. Our larger three benchmark programs are commonly used in evaluating pointer analyses. All our experiments were performed using Oracle HotSpot JRE 1.6.0 on a Linux machine with Intel Xeon 2.13 GHz processors and 128 Gb RAM.

We omit the modified top-down version of connection analysis from further evaluation, as its performance is similar to the original top-down version and its precision is (provably, and experimentally confirmed) identical to our bottom-up version.

Precision. Following [14], we measure precision by the size of the connection sets of pointer variables at program points of interest. Each pair of variable and program point can be viewed as a separate *query* to the connection analysis. To obtain such queries, we chose the parallelism client proposed in the original work of [14], which demands the connection set of each dereferenced pointer variable in the program. In Java, this corresponds to variables of reference type that are dereferenced to access instance fields

	description	# of classes		# of methods		# of bytecodes	
		app only	total	app only	total	app only	total
grande2	Java Grande kernels	17	61	112	237	8,146	13,724
grande3	Java Grande large-scale applications	42	241	231	1,162	27,812	75,139
antlr	Parser and translator generator	116	358	1,167	2,400	128,684	186,377
weka	Machine-learning library for data-mining tasks	62	530	575	3,391	40,767	223,291
bloat	Java bytecode optimization and analysis tool	277	611	2,651	4,699	194,725	311,727

Table 1. Benchmark characteristics for reachable code. (Reachable methods computed by a static 0-CFA call-graph analysis.) The "total" columns report numbers for *all* reachable code, whereas the "app only" columns report numbers for only application code (excluding JDK library code).

	# of queries	Bottom-Up analysis						Top-Down analysis			
		summary computation		summary instantiation							
		time	memory	time	memory	# of abstract states		time	memory	# of abstract states	
						queries	total			queries	total
grande2	616	0.6 sec	78 Mb	0.9 sec	61 Mb	616	1,318	1 sec	37 Mb	616	3,959
grande3	4,236	43 sec	224 Mb	1:21 min	137 Mb	4,373	8,258	1:11 min	506 Mb	4,354	27,232
antlr	5,838	16 sec	339 Mb	30 sec	149 Mb	6,207	21,437	1:23 min	1.1 Gb	8,388	79,710
weka	2,205	46 sec	503 Mb	2:48 min	228 Mb	2,523	25,147	$>$ 6 hrs	26 Gb	5,694	688,957
bloat	10,237	3:03 min	573 Mb	30 min	704 Mb	36,779	131,665	$>$ 6 hrs	24 Gb	139,551	962,376

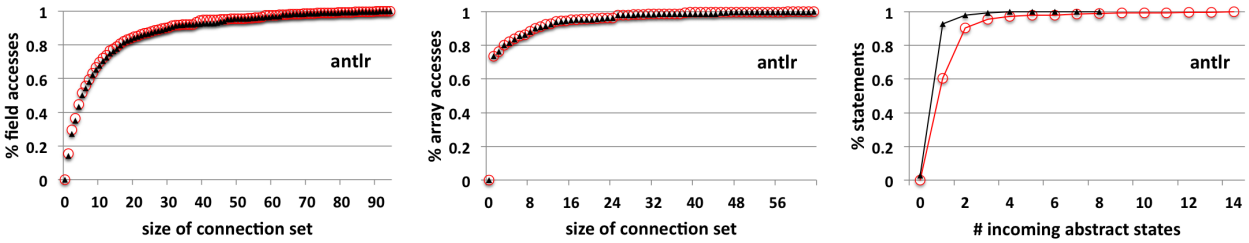
Table 2. A comparison of the scalability of the original top-down and our compositional bottom-up version. The measurements in the "query" sub-columns include only query points. The "total" sub-columns account for all points. All three metrics show that the top-down analysis scales much more poorly than the bottom-up analysis.

or array elements. More specifically, our queries constitute the base variable in each occurrence of a getfield, putfield, aload, or astore bytecode instruction in the program. The number of such queries for our five benchmarks are shown in the "# of queries" column of Tab. 2. To avoid counting the same set of queries across benchmarks, we only consider queries in application code, ignoring those in JDK library code. This number of queries ranges from around 0.6K to over 10K for our benchmarks.

Fig. 5 provides a detailed comparison of precision, based on the above metric, of the original top-down and bottom-up versions of connection analysis when applied to the antlr benchmark. Each graph in columns (a) and (b) plots, for each distinct connection set size (on the X axis), the fraction of queries (on the Y axis) for which each analysis computed connection sets of equal or smaller size. The graph shows that the precision of our modular bottom-up analysis closely tracks that of the original top-down analysis: the points for the bottom-up and top-down analyses, denoted ▲ and ○, respectively, overlap almost perfectly in each of the six graphs. The ratio of the connection set size computed by the top-down analysis to that computed by the bottom-up analysis on average across all queries is 0.952 for antlr (and 0.977 for grande2 and 0.977 for grande3). We do not, however, measure the impact of this precision loss of 2-5% on a real client. Note that for the largest two benchmarks, the top-down analysis timed-out.

Scalability. Tab. 2 compares the scalability of the top-down and bottom-up analyses in terms of three different metrics: running time, memory consumption, and the total number of computed abstract states. As noted earlier, the bottom-up analysis runs in two phases: a summary computation phase followed by a summary instantiation phase. The above data for these phases is reported in separate columns of the table. On our largest benchmark (bloat), the bottom-up analysis takes around 50 minutes and 873 Mb memory, whereas the top-down analysis times out after six hours, not only on this benchmark but also on the second largest one (weka).

The "# of abstract states" columns provide the sum of the sizes of the computed



(a) Precision of field accesses. (b) Precision of array accesses. (c) Scalability.

Fig. 5. Comparison of the precision and scalability of the original top-down and our modular bottom-up versions of connection analysis for the antlr benchmark.

abstractions in terms of the number of abstract states, including only incoming states at program points of queries (in the "queries" sub-column), and incoming states at all program points, including the JDK library (in the "total" sub-column). Column (c) of Fig. 5 provides more detailed measurements of the latter numbers. The graph shows, for each distinct number of incoming states computed at each program point (on the X axis), the fraction of program points (on the Y axis) with equal or smaller number of incoming states. The graphs clearly show the blow-up in the number of states computed by the top-down analysis over the bottom-up analysis.

7 Related Work, Discussion and Conclusions

The main technical observation in our work is that using right-modular abstract domains can help develop modular analyses. This observation is, in a way, similar to the frame rule in separation logic, in the sense that the join (resp. $*$) distributes over the transfer functions, and to the notion of *condensation* in logic programs [21].

The first compositional analysis framework was introduced in [6], and served as the basis for the concept of *abductive analysis* [15]. In [16], it has been shown that the semantic construction in [6] necessitates abstract domains which include functional objects and to a generalization of the reduced cardinal power domain [7] to arbitrary spaces of functions over a lattice. The latter was used to provide compositional semantics of logic programs [16]. These works paved the way to establishing the connection between modularity of analyses and *condensation* [17, 18].

Condensation is an algebraic property of abstract unification that ensures that it is possible to approximate the behavior of a query and then unify it with a given context and the obtained results are as precise as the ones obtained by analyzing the query after instantiating it in that specific context. Abstract domain which have this property are called condensing. Intuitively, in condensing domains it is possible to derive context-independent interprocedural (bottom-up) analyses with the same precision as the corresponding context-dependent (top-down) analysis. Examples for such domains are Boolean functions [24], Herbrand abstractions [17], equality based domain [25], or combinations of thereof [31]. (See [17] for further discussion.)

A lattice theoretic characterization of condensing abstract domains was suggested in [18] and later generalized in [17]. Intuitively, let C be an abstract domain, $S : C \to C$ the abstract semantics used in the analysis, and $\otimes$ an associate commutative binary operator $\otimes$, e.g., unification. S is said to be condensing for $\otimes$ if $S(a \otimes b) = a \otimes S(b)$

for any a and b in C [17, Def. 4.1]. In fact, a weaker characterization of condensation is given in [18] which, by using the notion of *weak completeness* [18, Def. 3.8] requires the equality to hold only for ordered pairs, i.e., when either $a \sqsubseteq b$ or $b \sqsubseteq a$ [18, The. 4.7]. However, when the order of the lattice is induced by the binary operator, as it is in our case where $\otimes = \sqcup$, the equality has to hold for any pair of elements. Thus, our requirements are less restrictive than theirs, as shown by the following example: Let $d_1 = d_2 = \{\{\bar{x}, x'\}, \{\bar{y}, y'\}\}$, then $[\![x = \texttt{null}]\!]^\sharp(d_1 \sqcup d_2) = \{\{\bar{x}\}, \{x'\}, \{\bar{y}, y'\}\}$ but $[\![x = \texttt{null}]\!]^\sharp(d_1) \sqcup d_2 = (d_1 \sqcap S_{x'}) \sqcup d_2) = \{\{\bar{x}, x'\}, \{\bar{y}, y'\}\}$.

The above example points to, what is arguably, the most subtle part of our work. Note that $d_2 \not\sqsubseteq S_{x'}$. Hence, although $[\![x = \texttt{null}]\!]^\sharp$ is conditionally adaptable (see Def. 3 and Lem. 1), we cannot take advantage of the modularity of $S_{x'}$. Surprisingly, we can benefit from the modularity of $S_{x'}$ when we adapt the result of the analysis of a procedure p (with ι_{entry} as input) to an arbitrary calling context. This is possible because the counterpart representation of calling contexts. Specifically, we can represent any calling context as a join between ι_{entry} and elements of the form $U_{\overline{xy}}$. Recall that $U_{\overline{xy}} = \{\{\bar{x}, \bar{y}\}\{x'\}\{y'\}$. Thus, it holds that $U_{\overline{xy}} \sqsubseteq S_{x'}$. In fact, for every x, y and z, it holds that $U_{\overline{xy}} \sqsubseteq S_{z'}$ (see §5.3.1).[4]

Conclusions. This paper shows that the notion of modularity from lattice theory can help for developing a precise bottom-up program analysis. In lieu of discussing the general framework [4], we illustrated the point by developing a compositional bottom-up connection analysis that has the same precision as the top-down counterpart, while enjoying the performance benefit of typically bottom-up analyses. Our analysis heavily uses modular elements in the abstract semantics of primitive commands, and their modularity property plays the key role in our proof that the precision of the compositional analysis coincides with that of the top-down counterpart. We also derived a new compositional analysis for a variant of the copy-constant propagation problem [13]. (See [4].) Our connection analysis can be used as a basis for a simple form of compositional taint analysis [22], essentially, by adding taint information to every partition. We hope that the connection we found between modularity in lattices and program analyses can help design precise and efficient compositional bottom-up analyses.

Acknowledgments. We thank the anonymous referees for their helpful comments. This work was supported by EU FP7 project ADVENT (308830), ERC grant agreement no. [321174-VSSC], Broadcom Foundation and Tel Aviv University Authentication Initiative, DARPA award #FA8750-12-2-0020 and NSF award #1253867.

References

1. Ball, T., Rajamani, S.: Bebop: a path-sensitive interprocedural dataflow engine. In: PASTE. (2001)
2. Bodden, E.: Inter-procedural data-flow analysis with ifds/ide and soot. In: SOAP. (2012)

[4] We note that if we only use elements in the analysis which are smaller than all the meet elements, then our analysis would become flow-insensitive. Let d, P_1, G_1, P_2, G_2 be abstract elements such that P_1 and P_2 are modular elements, $d \sqsubseteq P_1$, $((d \sqcap P_1) \sqcup G_1) \sqsubseteq P_2$, $((d \sqcap P_2) \sqcup G_2) \sqsubseteq P_1$. It holds that $G_1 \sqsubseteq P_2$, $G_2 \sqsubseteq P_1$, and thus $(((d \sqcap P_1) \sqcup G_1) \sqcap P_2) \sqcup G_2 = (((d \sqcap P_1) \sqcap P_2) \sqcup G_1) \sqcup G_2 = (d \sqcap P_2 \sqcap P_1) \sqcup G_2 \sqcup G_1 = ((d \sqcap P_2) \sqcup G_2) \sqcap P_1) \sqcup G_1$. (This is in fact the case if the meet elements are $\top$.) Note that this would imply that $[\![x = null]\!]^\sharp \circ [\![x = y]\!]^\sharp = [\![x = y]\!]^\sharp \circ [\![x = null]\!]^\sharp$, which is neither desired nor the case in our analysis.

3. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM (2011)
4. Castelnuovo, G.: Modular lattices for compositional interprocedural analysis. Master's thesis, School of Computer Science, Tel Aviv University (2012)
5. Chatterjee, R., Ryder, B.G., Landi, W.: Relevant context inference. In: POPL. (1999)
6. Codish, M., Debray, S., Giacobazzi, R.: Compositional analysis of modular logic programs. In: POPL. (1993)
7. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In: POPL. (1977)
8. Cousot, P., Cousot, R.: Static determination of dynamic properties of recursive procedures. In: Formal Descriptions of Programming Concepts. (1978)
9. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL. (1979)
10. Cousot, P., Cousot, R.: Modular static program analysis. In: CC. (2002)
11. Dillig, I., Dillig, T., Aiken, A., Sagiv, M.: Precise and compact modular procedure summaries for heap manipulating programs. In: PLDI. (2011)
12. Dolby, J., Fink, S., Sridharan, M.: T. J. Watson Libraries for Analysis (2006)
13. Fischer, C.N., Cytron, R.K., LeBlanc, R.J.: Crafting A Compiler. Addison-Wesley (2009)
14. Ghiya, R., Hendren, L.: Connection analysis: A practical interprocedural heap analysis for C. IJPP (1996)
15. Giacobazzi, R.: Abductive analysis of modular logic programs. JLP (1998)
16. Giacobazzi, R., Ranzato, F.: The reduced relative power operation on abstract domains. TCS (1999)
17. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract domains condensing. TOCL (2005)
18. Giacobazzi, R., Scozzari, F.: A logical model for relational abstract domains. TOPLAS (1998)
19. Gratzer, G.: General Lattice Theory. Birkhauser Verlag (1978)
20. Gulavani, B., Chakraborty, S., Ramalingam, G., Nori, A.: Bottom-up shape analysis using LISF. TOPLAS (2011)
21. Jacobs, D., Langen, A.: Static analysis of logic programs for independent and parallelism. JLP (1992)
22. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in java applications with static analysis. In: USENIX Security. (2005)
23. Madhavan, R., Ramalingam, G., Vaswani, K.: Purity analysis: An abstract interpretation formulation. In: SAS. (2011)
24. Marriott, K., Søndergaard, H.: Precise and efficient groundness analysis for logic programs. LOPLAS (1993)
25. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: POPL. (2004)
26. Naik, M.: Chord: A program analysis platform for Java (2006)
27. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL. (1995)
28. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. TCS (1996)
29. Salcianu, A., Rinard, M.: Purity and side effect analysis for Java programs. In: VMCAI. (2005)
30. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications. (1981)
31. Simon, A.: Deriving a complete type inference for hindley-milner and vector sizes using expansion. In: PEPM. (2013)
32. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for java programs. In: OOPSLA. (1999)